PROJECT 5A: FILE SYSTEM - READING

Minimum Requirements: 12 public tests

TEST DISTRIBUTION

Public tests – 16 tests | 77 points

Release tests – 0 tests | 0 points

Secret tests – 0 tests | 0 points

Yes, no secrets, you have all the test code to debug.

GET A NEW GEEKOS

There are some change in the geekos distribution, download a new geekos, it should also come with modified makefile and a .submit file.

WHAT'S EXPECTED

- The purpose of this project to add a writable filesystem, LFS, to GeekOS.
- Previous projects not required: can start with fresh build!
- Ensure Makefile.linux ties the second ide disk to a valid lfs image (which should already done).
- This is a simplified version of LFS and it is only used for GeekOS. It would be hard to find documentation anywhere else except the spec, so read the spec.

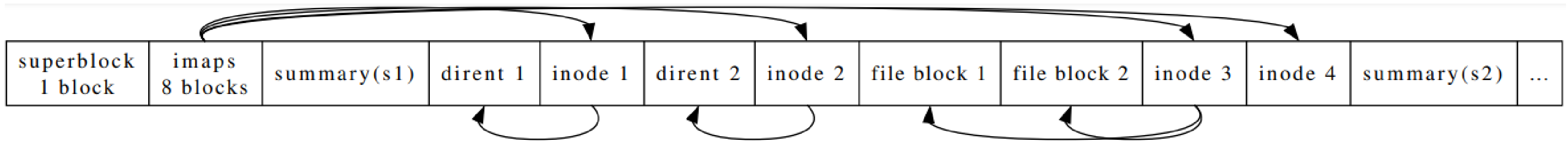
FUNCTIONS TO IMPLEMENT

- static int LFS_Mount(struct Mount_Point *mountPoint) //can test code after implementing this
- static int LFS_FStat(struct File *file, struct VFS_File_Stat *stat)
- static int LFS_Read(struct File *file, void *buf, ulong_t numBytes) // test 35 after this
- static int LFS_Seek(struct File *file, ulong_t pos)
- static int LFS_Close(struct File *file)
- static int LFS_FStat_Directory(struct File *dir, struct VFS_File_Stat *stat)
- static int LFS_Close_Directory(struct File *dir)
- static int LFS_Read_Entry(struct File *dir, struct VFS_Dir_Entry *entry)
- static int LFS_Open(struct Mount_Point *mountPoint, const char *path, int mode, struct File **pFile) // test 33
- static int LFS_Open_Directory(struct Mount_Point *mountPoint, const char *path, struct File **pDir)
- static int LFS_Stat(struct Mount_Point *mountPoint, const char *path, struct VFS_File_Stat *stat)

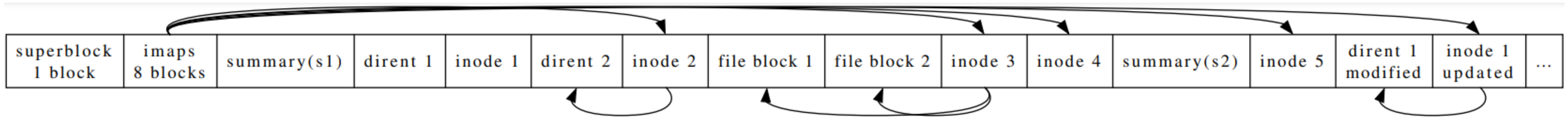
SYSTEM CALLS TO MODIFY

- Refer to the spec for details. You can go through Sys_Close, Sys_Open, Sys_Read , Sys_Write to see how they are implemented. There are some structures and functions you might want to refer to:
- Sys_Mount: call Mount(vfs.c) to mount a file system
- Sys_Open_Directory: next_descriptor, Open_Directory (vfs.c), add_file_to_descriptor_table
- Sys_ReadEntry: VFS_Dir_Entry, VFS_Dir_Entry
- Sys_Stat: VFS_File_Stat, Stat(vfs.c)
- Sys_FStat: VFS_File_Stat, FStat(vfs.c)
- Sys_Seek: Seek(vfs.c)

FILE SYSTEM & SEGMENTS

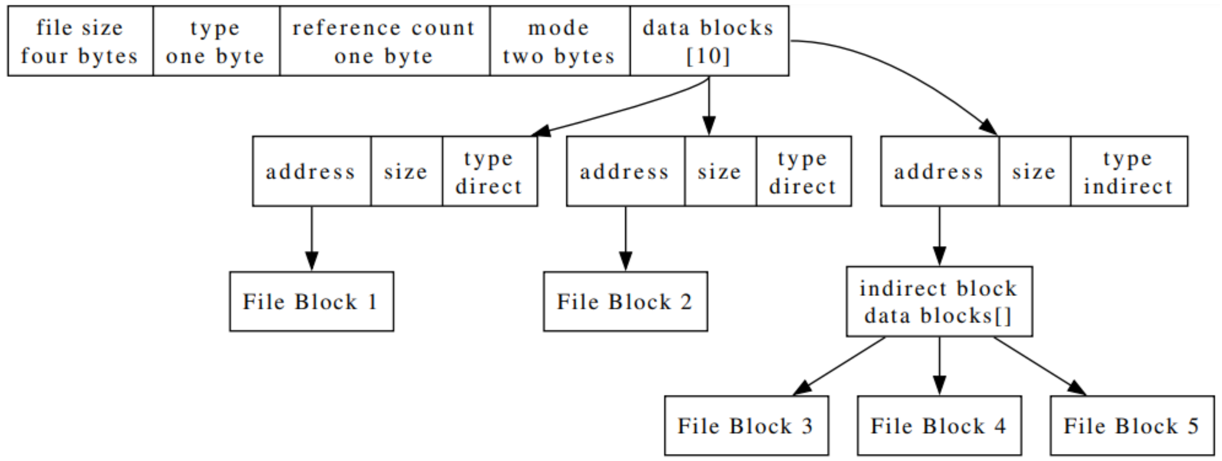


What happens if we create an empty file in the root:

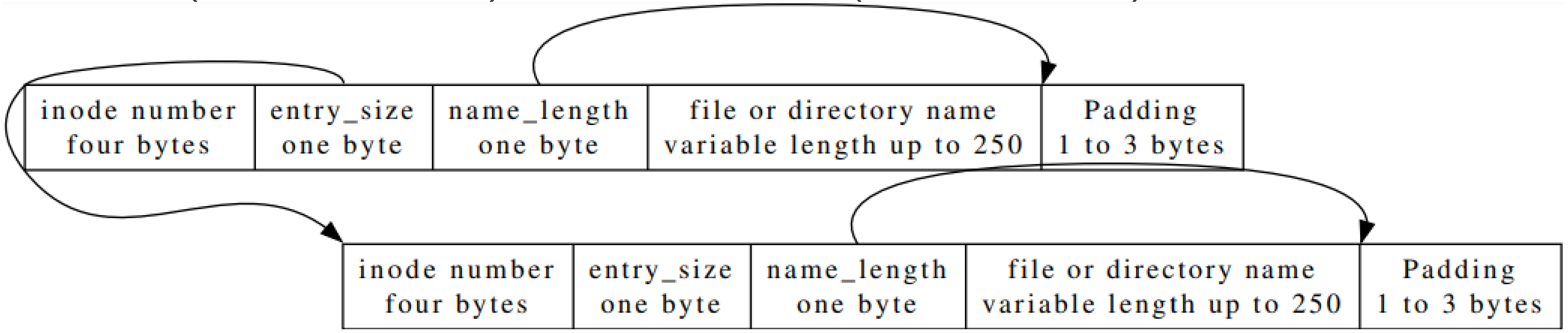


INODES

- DS that store information ABOUT file/directory
- Types: LFS_DIRECTORY or LFS_FILE. Inodes can be both directories or files.
- The extents points to the disk block which store the actual data of the file if the type is LFS_FILE but store dirents if the the type is LFS_DIRECTORY.
- File Blocks in an extent will be a contiguous memory
- Don't need to worry about mode.



DIRENT (LFS DIRENT) AND INODE (LFS INODE)



- Dirents are data structures used to store the contents of a directory. This is stored in the data blocks of a LFS DIRECTORY's inode. Shown as red arrows.
- Black arrows in dirents are not pointers, they are here only to illustrate the size.

LFS_MOUNT

- See pfat.c
- Define the structure of LFS instance (In-memory information describing a mounted LFS filesystem) . Create and allocate a LFS instance, and store in mountpoint→fsdata.
- Hint: You need at least the superblock information in the instance and you can add whatever parameter you need for the project.
- To load superblock:
    - Load the boot block (first block), find the size of the block in fileio.h SECTOR_SIZE(Need this size to malloc a buffer to pass to Block_read). Use Block_Read in blockdev.c.
    - Load the superblock from the boot block. It should be stored at byte offset LFS_BOOT_RECORD_OFFSET from the boot sector (use memcpy)
- Check the magic number of the superblock. The magic number will change if corrupted, will be 0 if not initialized (LFS_Magic)

LFS_MOUNT

- For cache(check struct “FS_Buffer_Cache”) , you can use Create_FS_Buffer_Cache.
- Note you should always use bufcache.h API to read from disk except maybe for the boot block.
- Store the lfs instance in mountpoint→fsdata.
- Store mountpoint → ops properly using the mount point ops defined for you.

LFS_OPEN

- Goal: Open a file given a path.
- e.g. : /home/test/file.c ← path
- You can get the instance of LFS_Instance from mountPoint→fsData where you stored earlier in LFS_Mount.
- Get inode number of the inode that contains the file
- Helpful to have the root directory inode as one of the fields in struct LFS_Instance
- More details on next slide

LFS_OPEN

- Split path into segments by '/'
- Loop through dirents under current directory (access dirents through data blocks in inode)
- If the dirent name matches the segment (e.g. 'home')
    - If the segment is not the last segment
        - Repeat the process for the found dirent corresponding to the desired subdirectory (e.g. 'home')
    - If the segment is the last segment (e.g. test.c):
        - Create a struct lfs_file (custom struct you should define, which stores all information to manipulate a file - inode and inode number)
        - Allocate struct File (Allocate_File() in vfs.c)
        - Store the file inode in the struct lfs_file, and set file→fsdata to point to your lfs_file
        - Store the struct File pointer you allocated to *pFile (you need to dereference the double pointer, it's changing the value of a pointer)

LFS_READ

- Goal: Reads data from the current position in file
- Parameters : struct File *file, void *buf, ulong_t numBytes
- Get the start pos(file→filePos) and calculate the end position.
- Retrieve the LFS file and the LFS instance you stored earlier in File→fsdata and File→mountpoint→fsdata.
- Get the inode from the LFS file (If you are not storing the inode in LFS file, then get the inode using the inode num)
- Walk through the data blocks and get the block for the start position
- It may be possible that file data is stored in more than one data block so you may have to read from multiple data blocks. For each data block, read proper size of data from the file into the buffer.
- Return the number of bytes read (Could make it to a helper function)

LFS_READ_ENTRY

- Goal: Read a directory entry from an open directory
- To read all files under a directory, you should keep calling the function until it returns VFS_NO_MORE_DIR_ENTRIES
- So you want to keep your progress across calls to read_entry (by using file→filepos). When it cannot find more entries, should return VFS_NO_MORE_DIR_ENTRIES
- Steps :
  - Retrieve the LFS file and the LFS instance (same file→fsdata, file→mountpoint→fsdata)
  - Get the inode from the struct LFS file (here file is a directory)
  - Loop through dirents making use of entry_length(check struct LFS_dirent)
  - Skip dirents until you hit the dirent after filepos
  - Increment filepos after that
  - Copy the name of the dirent to the VFS_Dir_Entry

LFS_STAT & LFS_FSTAT

- Get metadata of file named by given path or metadata for given file
- LFS_Stat :
  - You can get LFS_Instance from mountPoint→fsData (remember we stored it in LFS_mount)
  - Get the inode from the path (like you did in LFS_open)
- LFS_FStat :
  - You can get LFS_File from file→fsData
  - Get the inode of the file
- It should be easy if you have implemented the helper function of getting an inode
- Set VFS_File_Stat *stat correctly

LFS_FSTAT_DIRECTORY

- Goal : Stat operation for an already open directory
- You can get LFS_File from dir→fsData
- Get the inode of the directory
- Set VFS_File_Stat *stat correctly

LFS_OPEN_DIRECTORY

- Similar with LFS_Open, but be careful with the parameters in Allocate_File (mainly pass the proper mode)

GFS3_SEEK

- Parameter: struct File *file, ulong_t pos
- Set up filePos of a file

LFS_CLOSE & LFS_CLOSE_DIRECTORY

- Goal : Close a file or a directory
- Free all the resources you’ve allocated
- I.e. Free the LFS File

GFS3_DISK_PROPERTIES

- Use mountPoint to retrieve your LFS file system instance
- Fill in block_size and blocks_in_disk(can get this info from superblock in lfs instance)

NOTE

- Remember to do sanity checks and free memory when necessary.
- Use Release_FS_Buffer to release a buffer after using one, even if you're just reading it.
- Find root inode: Use ROOT_INODE to get the inode number of the root
- Create 2 struct - lfs_file and lfs_instance
- Check Mount_Point struct , Block_Read function
- The 0th inode is at the very beginning of block 1. The inodes address are like an array in the disk, the next inode address is just after the previous one.
- In the Dirent diagram in the spec, entry_size also includes the size of this field itself.
- Test Files to run – lfstst.c